

ATTORNEY DOCKET No.

NVIDP021/P000174

U.S. PATENT APPLICATION
FOR
SYSTEM, METHOD AND ARTICLE OF
MANUFACTURE FOR A PROGRAMMABLE
VERTEX PROCESSING MODEL WITH
INSTRUCTION SET

INVENTORS: John Erik Lindholm
Henry P. Moreton
Simon Moy
David B. Kirk

ASSIGNEE: **nVIDIA** CORPORATION

KEVIN J. ZILKA
PATENT AGENT
P.O. Box 721030
SAN JOSE, CA 95172

SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR A
PROGRAMMABLE VERTEX PROCESSING MODEL WITH
INSTRUCTION SET

By Inventors

John Erik Lindholm

Henry P. Moreton

Simon Moy

David B. Kirk

RELATED APPLICATION

The present application is a continuation-in-part of an application entitled
“METHOD, APPARATUS AND ARTICLE OF MANUFACTURE FOR A
TRANSFORM MODULE IN A GRAPHICS PROCESSOR” filed December 06,
1999 under serial number 09/456,102 and attorney docket number
NVIDP010/P000127 which is incorporated herein by reference in its entirety.

FIELD OF THE INVENTION

The present invention relates to computer graphics, and more particularly to
providing programmability in a computer graphics processing pipeline.

BACKGROUND OF THE INVENTION

Graphics application program interfaces (API's) have been instrumental in
allowing applications to be written to a standard interface and to be run on multiple

platforms, i.e. operating systems. Examples of such graphics API's include Open Graphics Library (OpenGL[®]) and D3D[™] transform and lighting pipelines.

OpenGL[®] is the computer industry's standard graphics API for defining 2-D and 3-D graphic images. With OpenGL[®], an application can create the same effects in any operating system using any OpenGL[®]-adhering graphics adapter. OpenGL[®] specifies a set of commands or immediately executed functions. Each command directs a drawing action or causes special effects.

Thus, in any computer system which supports this OpenGL[®] standard, the operating system(s) and application software programs can make calls according to the standard, without knowing exactly any specifics regarding the hardware configuration of the system. This is accomplished by providing a complete library of low-level graphics manipulation commands, which can be used to implement graphics operations.

A significant benefit is afforded by providing a predefined set of commands in graphics API's such as OpenGL[®]. By restricting the allowable operations, such commands can be highly optimized in the driver and hardware implementing the graphics API. On the other hand, one major drawback of this approach is that changes to the graphics API are difficult and slow to be implemented. It may take years for a new feature to be broadly adopted across multiple vendors.

With the impending integration of transform operations into high speed graphics chips and the higher integration levels allowed by semiconductor manufacturing, it is now possible to make part of the geometry pipeline accessible to the application writer. There is thus a need to exploit this trend in order to afford increased flexibility in visual effects. In particular, there is a need to provide a new computer graphics programming model and instruction set that allows convenient implementation of changes to the graphics API, while preserving the driver and hardware optimization afforded by currently established graphics API's.

DISCLOSURE OF THE INVENTION

5 A system, method and article of manufacture are provided for programmable processing in a computer graphics pipeline. Initially, data is received from a source buffer. Thereafter, programmable operations are performed on the data in order to generate output. The operations are programmable in that a user may utilize instructions from a predetermined instruction set for generating the same. Such
10 output is stored in a register. During operation, the output stored in the register is used in performing the programmable operations on the data.

 By this design, the present invention allows a user to program a portion of the graphics pipeline that handles vertex processing. This results in an increased
15 flexibility in generating visual effects. Further, the programmable vertex processing of the present invention allows remaining portions of the graphics pipeline, i.e. primitive processing, to be controlled by a standard graphics application program interface (API) for the purpose of preserving hardware optimizations.

20 In one embodiment of the present invention, only one vertex is processed at a time in a functional module that performs the programmable operations. Further, the various foregoing operations may be processed for multiple vertices in parallel.

 In another embodiment of the present invention, the data may include a
25 constant and/or vertex data. During operation, the constant may be stored in a constant source buffer and the vertex data may be stored in a vertex source buffer. Further, the constant may be accessed in the constant source buffer using an absolute or relative address.

30 In still another embodiment of the present invention, the register may be equipped with single write and triple read access. The output may also be stored in a

destination buffer. The output may be stored in the destination buffer under a predetermined reserved address.

As an option, the programmable vertex processing of the present invention
5 may include negating the data. Still yet, the programmable vertex processing may also involve swizzling the data. Data swizzling is useful when generating vectors. Such technique allows the efficient generation of a vector cross product and other vectors.

10 During operation, the programmable vertex processing is adapted for carrying out various instructions of an instruction set. Such instructions may include, but are not limited to a no operation, address register load, move, multiply, addition, multiply and addition, reciprocal, reciprocal square root, three component dot product, four component dot product, distance vector, minimum, maximum, set
15 on less than, set on greater or equal than, exponential base two (2), logarithm base two (2), and/or light coefficients.

These various instructions may each be carried out using a unique associated method and data structure. Such data structure includes a source location identifier
20 indicating a source location of data to be processed. Such source location may include a plurality of components. Further provided is a source component identifier indicating in which of the plurality of components of the source location the data resides. The data may be retrieved based on the source location identifier and the source component identifier. This way, the operation associated with the instruction
25 at hand may be performed on the retrieved data in order to generate output.

Also provided is a destination location identifier for indicating a destination location of the output. Such destination location may include a plurality of components. Further, a destination component identifier is included indicating in
30 which of the plurality of components of the destination location the output is to be

stored. In operation, the output is stored based on the destination location identifier and the destination component identifier.

These and other advantages of the present invention will become apparent upon
5 reading the following detailed description and studying the various figures of the
drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

5 The foregoing and other aspects and advantages are better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

10 Figure 1 is a conceptual diagram illustrating a graphics pipeline in accordance with one embodiment of the present invention;

15 Figure 2 illustrates the overall operation of the various components of the graphics pipeline of Figure 1;

20 Figure 3 is a schematic illustrating one embodiment of a programming model in accordance with the present invention;

25 Figure 4 is a flowchart illustrating the method by which the programming model of Figure 3 carries out programmable vertex processing in the computer graphics pipeline; and

30 Figure 5 is a flowchart illustrating the method in a data structure is employed to carry out graphics instructions in accordance with one embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 is a conceptual diagram illustrating a graphics pipeline 100 in accordance with one embodiment of the present invention. During use, the graphics pipeline 100 is adapted to carry out numerous operations for the purpose of processing computer graphics. Such operations may be categorized into two types, namely vertex processing 102 and primitive processing 104. At least partially during use, the vertex processing 102 and primitive processing 104 adhere to a standard graphics application program interface (API) such as OpenGL® or any other desired graphics API.

Vertex processing 102 normally leads primitive processing 104, and includes well known operations such as texgen operations, lighting operations, transform operations, and/or any other operations that involve vertices in the computer graphics pipeline 100.

Primitive processing 104 normally follows vertex processing 102, and includes well known operations such as culling, frustum clipping, polymode operations, flat shading, polygon offsetting, fragmenting, and/or any other operations that involve primitives in the computer graphics pipeline 100. It should be noted that still other operations may be performed such as viewport operations.

Figure 2 illustrates a high level operation 200 of the graphics pipeline 100 of Figure 1. As shown, it is constantly determined in decision 202 whether current operation invokes a programmable geometry model of the present invention. If so, a mode is enabled that partially supercedes the vertex processing 102 of the standard graphics API, thus providing increased flexibility in generating visual effects. See operation 204.

30

When disabled, the present invention allows increased or exclusive control of the graphics pipeline 100 by the standard graphics API, as indicated in operation 206. In one embodiment, states of the standard graphics API state may not be overruled by invoking the programmable geometry mode of the present invention.

5 In one embodiment, no graphics API state may be directly accessible by the present invention.

In one embodiment of the present invention, the programmable geometry mode of the present invention may optionally be limited to vertex processing from object space into homogeneous clip space. This is to avoid compromising hardware performance that is afforded by allowing exclusive control of the primitive processing 104 by the standard graphics API at all times.

10

The remaining description will be set forth assuming that the programmable geometry mode supersedes the standard graphics API only during vertex processing 102. It should be noted, however, that in various embodiments of the present invention, the programmable geometry mode may also supersede the standard graphics API during primitive processing 104.

15

Figure 3 is a schematic illustrating one embodiment of a programming model 300 in accordance with the present invention. Such programming model 300 may be adapted to work with hardware accelerators of various configuration and/or with central processing unit (CPU) processing.

20

As shown in Figure 3, the programming module 300 includes a functional module 302 that is capable of carrying out a plurality of different types of operations. The functional module 302 is equipped with three inputs and an output. Associated with each of the three inputs is a swizzling module 304 and a negating module 306 for purposes that will be set forth hereinafter in greater detail.

25

30

Coupled to the output of the functional module 302 is an input of a register 308 having three outputs. Also coupled to the output of the functional module 302 is a vertex destination buffer 310. The vertex destination buffer 310 may include a vector component write mask, and may preclude read access.

5

Also included are a vertex source buffer 312 and a constant source buffer 314. The vertex source buffer 312 stores data in the form of vertex data, and may be equipped with write access and/or at least single read access. The constant source buffer 314 stores data in the form of constant data, and may also be equipped with write access and/or at least single read access.

10

Each of the inputs of the functional module 302 is equipped with a multiplexer 316. This allows the outputs of the register 308, vertex source buffer 312, and constant source buffer 314 to be fed to the inputs of the functional module 302. This is facilitated by buses 318.

15

Figure 4 is a flowchart illustrating the method 400 by which the model of Figure 3 carries out programmable vertex processing in the computer graphics pipeline 100. Initially, in operation 402, data is received from a vertex source buffer 312. Such data may include any type of information that is involved during the processing of vertices in the computer graphics pipeline 100. Further, the vertex source buffer 312 may include any type of memory capable of storing data.

20

Thereafter, in operation 404, programmable operations, i.e. vertex processing 102, are performed on the data in order to generate output. The programmable operations are capable of generating output including at the very least a position of a vertex in homogeneous clip space. In one embodiment, such position may be designated using Cartesian coordinates each with a normalized range between -1.0 and 1.0. Such output is stored in the register 308 in operation 406. During operation 408, the output stored in the register 308 is used in performing the programmable

25

30

operations on the data. Thus, the register 308 may include any type of memory capable of allowing the execution of the programmable operations on the output.

By this design, the present invention allows a user to program a portion of the graphics pipeline 100 that handles vertex processing. This results in an increased flexibility in generating visual effects. Further, the programmable vertex processing of the present invention allows remaining portions of the graphics pipeline 100 to be controlled by the standard application program interface (API) for the purpose of preserving hardware optimizations.

10

During operation, only one vertex is processed at a time in the functional module 302 that performs the programmable operations. As such, the vertices may be processed independently. Further, the various foregoing operations may be processed for multiple vertices in parallel.

15

In one embodiment of the present invention, a constant may be received, and the programmable operations may be performed based on the constant. During operation, the constant may be stored in and received from the constant source buffer 314. Further, the constant may be accessed in the constant source buffer 314 using an absolute or relative address. As an option, there may be one address register for use during reads from the constant source buffer 314. It may be initialized to 0 at the start of program execution in operation 204 of Figure 2. Further, the constant source buffer 314 may be written with a program which may or may not be exposed to users.

25

The register 308 may be equipped with single write and triple read access. Register contents may be initialized to (0,0,0,0) at the start of program execution in operation 204 of Figure 2. It should be understood that the output of the functional module 302 may also be stored in the vertex destination buffer 310. The vertex position output may be stored in the vertex destination buffer 310 under a predetermined reserved address. The contents of the vertex destination buffer 310

30

may be initialized to (0,0,0,1) at the start of program execution in operation 204 of Figure 2.

As an option, the programmable vertex processing may include negating the data. Still yet, the programmable vertex processing may also involve swizzling the data. Data swizzling is useful when generating vectors. Such technique allows the efficient generation of a vector cross product and other vectors.

In one embodiment, the vertex source buffer 312 may be 16 quad-words in size (16*128 bits). Execution of the present invention may be commenced when Param[0]/Position is written. All attributes may be persistent. That is, they remain constant until changed. Table 1 illustrates the framework of the vertex source buffer 312. It should be noted that the number of textures supported may vary across implementations.

Table 1

Program Mode	Standard API
Param[0] X,Y,Z,W	Position X,Y,Z,W
Param[1] X,Y,Z,W	Skin Weights W,W,W,W
Param[2] X,Y,Z,W	Normal X,Y,Z,*
Param[3] X,Y,Z,W	Diffuse Color R,G,B,A
Param[4] X,Y,Z,W	Specular Color R,G,B,A
Param[5] X,Y,Z,W	Fog F,*,*,*
Param[6] X,Y,Z,W	Point Size P,*,*,*
Param[7] X,Y,Z,W	*,*,*,*
Param[8] X,Y,Z,W	*,*,*,*
Param[9] X,Y,Z,W	Texture0 S,T,R,Q
Param[10] X,Y,Z,W	Texture1 S,T,R,Q
Param[11] X,Y,Z,W	Texture2 S,T,R,Q

Param[12]	X,Y,Z,W	Texture3	S,T,R,Q
Param[13]	X,Y,Z,W	Texture4	S,T,R,Q
Param[14]	X,Y,Z,W	Texture5	S,T,R,Q
Param[15]	X,Y,Z,W	Texture6	S,T,R,Q

In another embodiment, the vertex destination buffer **310** may be 13 quad-words in size and may be deemed complete when the program is finished. The following exemplary vertex destination buffer addresses are pre-defined to fit a standard pipeline. Contents are initialized to (0,0,0,1) at start of program execution in operation **204** of Figure 2. Writes to locations that are not used by the downstream hardware may be ignored.

A reserved address (HPOS) may be used to denote the homogeneous clip space position of the vertex in the vertex destination buffer **310**. It may be generated by the geometry program. Table 2 illustrates the various locations of the vertex destination buffer **310** and a description thereof.

Table 2

Location	Description
HPOS	HClip Position x,y,z,w (-1.0 to 1.0)
COL0	Color0 (diff) r,g,b,a (0.0 to 1.0)
COL1	Color1 (spec) r,g,b,a (0.0 to 1.0)
BCOL0	Color0 (diff) r,g,b,a (0.0 to 1.0)
BCOL1	Color1 (spec) r,g,b,a (0.0 to 1.0)
FOGP	Fog Parameter f,*,*,*
PSIZ	Point Size p,*,*,*
TEX0	Texture0 s,t,r,q
TEX1	Texture1 s,t,r,q

TEX2	Texture2	s,t,r,q
TEX3	Texture3	s,t,r,q

5 HPOS - homogeneous clip space position
float[4] x,y,z,w
- standard graphics pipeline process further
(clip check, perspective divide, viewport
scale and bias).

10 COL0/BCOL0 - color0 (diffuse)
COL1/BCOL1 - color1 (specular)
float[4] r,g,b,a
- each component gets clamped to (0.0,1.0)
before interpolation

15 - each component is interpolated at least as
8-bit unsigned integer.

 TEX0-7 - textures 0 to 7
float[4] s,t,r,q
- each component is interpolated as high
precision float, followed by division of q
and texture lookup. Extra colors could use
texture slots. Advanced fog can be done as a
texture.

25 FOGP fog parameter
float[1] f (distance used in fog equation)
- gets interpolated as a medium precision
float and used in a fog evaluation (linear,
exp, exp2) generating a fog color blend
value.

30 PSIZ point size
float[1] p
- gets clamped to (0.0,POINT_SIZE_MAX) and
used as point size.

35

An exemplary assembly language that may be used in one implementation of the present invention will now be set forth. In one embodiment, no branching instructions may be allowed for maintaining simplicity. It should be noted, however, that branching may be simulated using various combinations of operations, as is well known to those of ordinary skill. Table 3 illustrates a list of the various resources associated with the programming model 300 of Figure 3. Also shown in a reference format associated with each of the resources along with a proposed size thereof.

Table 3

10

Resources:

	Vertex Source	- v[*]	of size 16 vectors (256B)
	Constant Memory	- c[*]	of size 192 vectors (1536B)
15	Address Register	- A0.x	of size 1 signed integer (or multiple vectors)
	Data Registers	- R0-R11,R12	of size 13 vectors (192B)
	Vertex Destination	- o[*]	of size 11 vectors (208B)
20	Instruction Storage		of size 128 instructions

Note: All data registers and memory locations may be four component floats.

25

For example, the constant source buffer 314 may be accessed as c[*] (absolute) or as c[A0.x+*] (relative). In the relative case, a 32-bit signed address register may be added to the read address. Out of range address reads may result in (0,0,0,0). In one embodiment, the vertex source buffer 312, vertex destination buffer 310, and register 308 may not use relative addressing.

30

Vector components may be swizzled before use via four subscripts (xyzw). Accordingly, an arbitrary component re-mapping may be done. Examples of swizzling commands are shown in Table 4.

Table 4

5 .xyzw means source(x,y,z,w) → input(x,y,z,w)
 .zzxy means source(x,y,z,w) → input(z,z,x,y)
 .xxxx means source(x,y,z,w) → input(x,x,x,x)

Table 5 illustrates an optional shortcut notation of the assembly language that may be permitted.

Table 5

10

15

No subscripts is the same as .xyzw
.x is the same as .xxxx
.y is the same as .yyyy
.z is the same as .zzzz
.w is the same as .www

20 All source operands may be negated by putting a '-' sign in front of the above notation. Writes to the register **308** may be maskable. In other words, each component may be written only if it appears as a destination subscript (from xyzw). No swizzling may be possible for writes, and subscripts may be ordered (x before y before z before w).

25 Writes to the vertex destination buffer **310** and/or the constant memory **314** may also be maskable. Each component may be written only if it appears as a destination subscript (from xyzw). No swizzling may be permitted for writes, and subscripts may be ordered (x before y before z before w).

30 An exemplary assembler format is as follows:

OPCODE DESTINATION, SOURCE(S)

Generated data may be written to the register **308** or the vertex destination buffer **310**. Output data is taken from the functional module **302**. Table 6 illustrates commands in the proposed assembler format which write output to the register **308** or the vertex destination buffer **310**.

5

Table 6

ADD R4,R1,R2 result goes into R4
ADD o[HPOS],R1,R2 result goes into the destination buffer
10 ADD R4.xy,R1,R2 result goes into x,y components of R4

During operation, the programmable vertex processing is adapted for carrying out various instructions of an instruction set using any type of programming language including, but not limited to that set forth hereinabove. Such instructions
15 may include, but are not limited to a no operation, address register load, move, multiply, addition, multiply and addition, reciprocal, reciprocal square root, three component dot product, four component dot product, distance vector, minimum, maximum, set on less than, set on greater or equal than, exponential base two (2), logarithm base two (2), and/or light coefficients. Table 7 illustrates the operation
20 code associated with each of the foregoing instructions. Also indicated is a number of inputs and outputs as well as whether the inputs and outputs are scalar or vector.

Table 7

OPCODE	INPUT(scalar or vector)	OUTPUT(replicated scalar or vector)
NOP		
ARL	s	
MOV	v	v
MUL	v,v	v
ADD	v,v	v
MAD	v,v,v	v
RCP	s	s,s,s,s or v or v or v
RSQ	s	s,s,s,s or v

DP3	v,v	s,s,s,s
DP4	v,v	s,s,s,s
DST	v,v	V
MIN	v,v	V
MAX	v,v	V
SLT	v,v	V
SGE	v,v	V
EXP	s	V
LOG	s	v
LIT	v	v

As shown in Table 7, each of the instructions includes an input and an output which may take the form of a vector and/or a scalar. It should be noted that such vector and scalar inputs and outputs may be handled in various ways. Further information on dealing with such inputs and outputs may be had by reference to a co-pending application entitled "METHOD, APPARATUS AND ARTICLE OF MANUFACTURE FOR A TRANSFORM MODULE IN A GRAPHICS PROCESSOR" filed December 06, 1999 under serial number 09/456,102 and attorney docket number NVIDP010/P000127 which is incorporated herein by reference in its entirety.

These various instructions may each be carried out using a unique associated method and data structure. Such data structure includes a source location identifier indicating a source location of data to be processed. Such source location may include a plurality of components. Further provided is a source component identifier indicating in which of the plurality of components of the source location the data resides. The data may be retrieved based on the source location identifier and the source component identifier. This way, the operation associated with the instruction at hand may be performed on the retrieved data in order to generate output.

20

Also provided is a destination location identifier for indicating a destination location of the output. Such destination location may include a plurality of components. Further, a destination component identifier is included indicating in which of the plurality of components of the destination location the output is to be

stored. In operation, the output is stored based on the destination location identifier and the destination component identifier.

Figure 5 is a flowchart illustrating the method 500 in which the foregoing data structure is employed in carrying out the instructions in accordance with one embodiment of the present invention. First, in operation 502, the source location identifier is received indicating a source location of data to be processed. Thereafter, in operation 504, the source component identifier is received indicating in which of the plurality of components of the source location the data resides.

10

The data is subsequently retrieved based on the source location identifier and the source component identifier, as indicated in operation 506. Further, the particular operation is performed on the retrieved data in order to generate output. See operation 508. The destination location identifier is then identified in operation 510 for indicating a destination location of the output. In operation 512, the destination component identifier is identified for indicating in which of the plurality of components of the destination location the output is to be stored. Finally, in operation 514, the output is stored based on the destination location identifier and the destination component identifier.

20

Further information will now be set forth regarding each of the instructions set forth in Table 7. In particular, an exemplary format, description, operation, and examples are provided using the programming language set forth earlier.

25 Address Register Load (ARL)

Format:

ARL A0.x,[-]S0.[xyzw]

30

Description:

The contents of source scalar are moved into a specified address register. Source may have one subscript. Destination may have an “.x” subscript. In one embodiment, the only valid address register may be designated as “A0.x.” The address register “A0.x” may be used as a base address for constant reads. The source may be a float that is truncated towards negative infinity into a signed integer.

Operation:

Table 8A sets forth an example of operation associated with the ARL instruction.

Table 8A

```
t.x = source0.c***; /* c is x or y or z or w */
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (-source0)
    t = -t;

A0.x = TruncateTo-Infinity(t.x);
```

Examples:

ARL A0.x,v[7].w (move vertex scalar into address register 0)
MOV R6,c[A0.x+7] (move constant at address A0.x+7 into register R6)

Mov (MOV)

Format:

MOV D[.xyzw],[**-**]S0[.xyzw]

Description:

The contents of a designated source are moved into a destination.

Operation:

5

Table 8B sets forth an example of operation associated with the MOV instruction.

Table 8B

10

```
t.x = source0.c***;  
t.y = source0.*c**;  
t.z = source0.**c*;  
t.w = source0.***c;  
15 if (negate0) {  
    t.x = -t.x;  
    t.y = -t.y;  
    t.z = -t.z;  
    t.w = -t.w;  
20 }  
if (xmask) destination.x = t.x;  
if (ymask) destination.y = t.y;  
if (zmask) destination.z = t.z;  
25 if (wmask) destination.w = t.w;
```

Examples:

```
MOV o[1],-R4 (move negative R4 into o[1])  
30 MOV R5,v[POS].w (move w component of v[POS] into xyzw components of R5)  
MOV o[HPOS],c[0] (output constant in location zero)  
MOV R7.xyw,R4.x (move x component of R4 into x,y,w components of R7)
```

Multiply (MUL)

35

Format:

MUL D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

Description:

The present instruction multiplies sources into a destination. It should be noted that 0.0 times anything is 0.0.

5

Operation:

Table 8C sets forth an example of operation associated with the MUL instruction.

10

Table 8C

15

20

25

30

35

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
    t.y = -t.y;
    t.z = -t.z;
    t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
    u.w = -u.w;
}
if (xmask) destination.x = t.x * u.x;
if (ymask) destination.y = t.y * u.y;
if (zmask) destination.z = t.z * u.z;
if (wmask) destination.w = t.w * u.w;
```

Examples:

40

```
MUL R6,R5,c[CON5] R6.xyzw = R5.xyzw * c[CON5].xyzw
MUL R6.x,R5.w,-R7 R6.x = R5.w*-R7.x
```

Add (ADD)

Format:

ADD D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

5

Description:

The present instruction adds sources into a destination.

10 Operation:

Table 8D sets forth an example of operation associated with the ADD instruction.

15

Table 8D

20

25

30

35

40

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
    t.y = -t.y;
    t.z = -t.z;
    t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
    u.w = -u.w;
}
if (xmask) destination.x = t.x + u.x;
if (ymask) destination.y = t.y + u.y;
if (zmask) destination.z = t.z + u.z;
if (wmask) destination.w = t.w + u.w;
```

Examples:

ADD R6,R5.x,c[CON5] R6.xyzw = R5.x + c[CON5].xyzw
ADD R6.x,R5,-R7 R6.x = R5.x - R7.x
ADD R6,-R5,c[CON5] R6.xyzw = -R5.xyzw + c[CON5].xyzw

5

Multiply And Add (MAD)

Format:

10

MAD D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw],[-]S2[.xyzw]

Description:

15

The present instruction multiplies and adds sources into a destination. It should be noted that 0.0 times anything is 0.0.

Operation:

20

Table 8E sets forth an example of operation associated with the MAD instruction.

Table 8E

25

```
t.x = source0.c***;  
t.y = source0.*c**;  
t.z = source0.**c*;  
t.w = source0.***c;  
if (negate0) {  
    t.x = -t.x;  
    t.y = -t.y;  
    t.z = -t.z;  
    t.w = -t.w;  
}  
u.x = source1.c***;  
u.y = source1.*c**;  
u.z = source1.**c*;
```

30

35


```

5      u.w = source1.***c;
      if (negate1) {
          u.x = -u.x;
          u.y = -u.y;
          u.z = -u.z;
          u.w = -u.w;
      }
10     v.x = source2.c***;
      v.y = source2.*c**;
      v.z = source2.**c*;
      v.w = source2.***c;
      if (negate2) {
15         v.x = -v.x;
         v.y = -v.y;
         v.z = -v.z;
         v.w = -v.w;
      }
20     if (xmask) destination.x = t.x * u.x + v.x;
      if (ymask) destination.y = t.y * u.y + v.y;
      if (zmask) destination.z = t.z * u.z + v.z;
      if (wmask) destination.w = t.w * u.w + v.w;
```

Examples:

MAD R6,-R5,v[POS],-R3 R6 = -R5 * v[POS] - R3

MAD R6.z,R5.w,v[POS],R5 R6.z = R5.w * v[POS].z + R5.z

Reciprocal (RCP)

Format:

RCP D[.xyzw],[-]S0.[xyzw]

Description:

The present instruction inverts a source scalar into a destination. The source may have one subscript. Output may be exactly 1.0 if the input is exactly 1.0.

RCP(-Inf) gives (-0.0,-0.0,-0.0,-0.0)

RCP(-0.0) gives (-Inf,-Inf,-Inf,-Inf)

RCP(+0.0) gives (+Inf,+Inf,+Inf,+Inf)

RCP(+Inf) gives (0.0,0.0,0.0,0.0)

Operation:

5

Table 8F sets forth an example of operation associated with the RCP instruction.

Table 8F

10

```
t.x = source0.c;
if (negate0) {
    t.x = -t.x;
}
if (t.x == 1.0f) {
    u.x = 1.0f;
} else {
    u.x = 1.0f / t.x;
}
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

15

20

25

where

$$| u.x - IEEE(1.0f/t.x) | < 1.0f/(2^{22})$$

30

for $1.0f \leq t.x \leq 2.0f$. The intent of this precision requirement is that this amount of relative precision apply over all values of $t.x$.

35 Examples:

RCP R2,c[A0.x+14].x R2.xyzw = 1/c[A0.x+14].x

RCP R2.w,R3.z R2.w = 1/R3.z

40 **Reciprocal Square Root (RSQ)**

Format:

RSQ D[.xyzw],[-]S0.[xyzw]

Description:

5

The present instruction performs an inverse square root of absolute value on a source scalar into a destination. The source may have one subscript. The output may be exactly 1.0 if the input is exactly 1.0.

10

RSQ(0.0) gives (+Inf,+Inf,+Inf,+Inf)

RSQ(Inf) gives (0.0,0.0,0.0,0.0)

Operation:

15

Table 8G sets forth an example of operation associated with the RSQ instruction.

Table 8G

20

```
t.x = source0.c;
if (negate0) {
    t.x = -t.x;
}
if (fabs(t.x) == 1.0f) {
    u.x = 1.0f;
} else {
    u.x = 1.0f / sqrt(fabs(t.x));
}
if (xmask) destination.x = u.x;
if (ymask) destination.y = u.x;
if (zmask) destination.z = u.x;
if (wmask) destination.w = u.x;
```

25

30

35

where

| u.x - IEEE(1.0f/sqrt(fabs(t.x))) | < 1.0f/(2²²)

40

for 1.0f <= t.x <= 4.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

Examples:

RSQ o[PA0],R3.y o[PA0] = 1/sqrt(abs(R3.y))

RSQ R2.w,v[9].x R2.w = 1/sqrt(abs(v[9].x))

5

Three Component Dot Product (DP3)

Format:

10 DP3 D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

Description:

15 The present instruction performs a three component dot product of the sources into a destination. It should be noted that 0.0 times anything is 0.0.

Operation:

20 Table 8H sets forth an example of operation associated with the DP3 instruction.

Table 8H

```
25      t.x = source0.c***;  
      t.y = source0.*c**;  
      t.z = source0.**c*;  
      if (negate0) {  
30          t.x = -t.x;  
          t.y = -t.y;  
          t.z = -t.z;  
      }  
      u.x = source1.c***;  
      u.y = source1.*c**;  
      u.z = source1.**c*;  
35      if (negate1) {  
          u.x = -u.x;  
          u.y = -u.y;  
          u.z = -u.z;
```

```

    }
    v.x = t.x * u.x + t.y * u.y + t.z * u.z;
    if (xmask) destination.x = v.x;
    if (ymask) destination.y = v.x;
    if (zmask) destination.z = v.x;
    if (wmask) destination.w = v.x;

```

Examples:

```

10      DP3 R6,R3,R4      R6.xyzw = R3.x*R4.x + R3.y*R4.y + R3.z*R4.z
      DP3 R6.w,R3,R4      R6.w = R3.x*R4.x + R3.y*R4.y + R3.z*R4.z

```

Four Component Dot Product (DP4)

15 Format:

DP4 D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

Description:

20

The present instruction performs a four component dot product of the sources into a destination. It should be noted that 0.0 times anything is 0.0.

Operation:

25

Table 8I sets forth an example of operation associated with the DP4 instruction.

Table 8I

30

```

    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
35    if (negate0) {
        t.x = -t.x;
        t.y = -t.y;

```

```

        t.z = -t.z;
        t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negatel) {
        u.x = -u.x;
        u.y = -u.y;
        u.z = -u.z;
        u.w = -u.w;
    }
    v.x = t.x * u.x + t.y * u.y + t.z * u.z + t.w * u.w;
    if (xmask) destination.x = v.x;
    if (ymask) destination.y = v.x;
    if (zmask) destination.z = v.x;
    if (wmask) destination.w = v.x;

```

20 Examples:

DP4 R6,v[POS],c[MV0] R6.xyzw = v.x*c.x + v.y*c.y + v.z*c.z +
v.w*c.w

DP4 R6.xw,v[POS].w,R3 R6.xw = v.w*R3.x + v.w*R3.y + v.w*R3.z +
v.w*R3.w

Distance Vector (DST)

Format:

30 DST D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

Description:

35 The present instruction calculates a distance vector. A first source vector is assumed to be (NA,d*d,d*NA) and a second source vector is assumed to be (NA,1/d,NA,1/d). A destination vector is then outputted in the form of (1,d,d,1/d). It should be noted that 0.0 times anything is 0.0.

40 Operation:

Table 8J sets forth an example of operation associated with the DST instruction.

5

Table 8J

```

t.y = source0.*c**;  

t.z = source0.**c**;  

10  if (negate0) {  

    t.y = -t.y;  

    t.z = -t.z;  

    }  

u.y = source1.*c**;  

u.w = source1.**c**;  

15  if (negate1) {  

    u.y = -u.y;  

    u.w = -u.w;  

    }  

20  if (xmask) destination.x = 1.0;  

    if (ymask) destination.y = t.y*u.y;  

    if (zmask) destination.z = t.z;  

    if (wmask) destination.w = u.w;
```

25 Examples:

DST R2,R3,R4 R2.xyzw = (1.0,R3.y*R4.y,R3.z,R4.w)

Minimum (MIN)

30

Format:

MIN D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

35 Description:

The present instruction determines a minimum of sources, and moves the same into a destination.

40 Operation:

Table 8K sets forth an example of operation associated with the MIN instruction.

5

Table 8K

10

15

20

25

30

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
    t.y = -t.y;
    t.z = -t.z;
    t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
    u.w = -u.w;
}
if (xmask) destination.x = (t.x < u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y < u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z < u.z) ? t.z : u.z;
if (wmask) destination.w = (t.w < u.w) ? t.w : u.w;
```

Examples:

35 MIN R2,R3,R4 R2 = component min(R3,R4)

MIN R2.x,R3.z,R4 R2.x = min(R3.z,R4.x)

Maximum (MAX)

40 Format:

MAX D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

Description:

The present instruction determines a maximum of sources, and moves the same into a destination.

5

Operation:

Table 8L sets forth an example of operation associated with the MAX instruction.

10

Table 8L

15

20

25

30

35

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
    t.y = -t.y;
    t.z = -t.z;
    t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
    u.w = -u.w;
}
if (xmask) destination.x = (t.x >= u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y >= u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z >= u.z) ? t.z : u.z;
if (wmask) destination.w = (t.w >= u.w) ? t.w : u.w;
```

Examples:

40

MAX R2,R3,R4 R2 = component max(R3,R4)
MAX R2.w,R3.x,R4 R2.w = max(R3.x,R4.w)

Set On Less Than (SLT)

Format:

SLT D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]

5

Description:

The present instruction sets a destination to 1.0/0.0 if source0 is less_than/greater_or_equal to source1. The following relationships should be noted:

10

SetEQ R0,R1 = (SGE R0,R1) * (SGE -R0,-R1)

SetNE R0,R1 = (SLT R0,R1) + (SLT -R0,-R1)

SetLE R0,R1 = SGE -R0,-R1

SetGT R0,R1 = SLT -R0,-R1

15

Operation:

Table 8M sets forth an example of operation associated with the SLT instruction.

20

Table 8M

25

```
t.x = source0.c***;
t.y = source0.*c**;
```

30

```
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
```

35

```
    t.y = -t.y;
    t.z = -t.z;
    t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
```

```

    u.x = -u.x;
    u.y = -u.y;
    u.z = -u.z;
    u.w = -u.w;
5      }
    if (xmask) destination.x = (t.x < u.x) ? 1.0 : 0.0;
    if (ymask) destination.y = (t.y < u.y) ? 1.0 : 0.0;
    if (zmask) destination.z = (t.z < u.z) ? 1.0 : 0.0;
10    if (wmask) destination.w = (t.w < u.w) ? 1.0 : 0.0;
```

Examples:

```

    SLT R4,R3,R7    R4.xyzw = (R3.xyzw < R7.xyzw ? 1.0 : 0.0)
15    SLT R3.xz,R6.w,R4  R3.xz = (R6.w < R4.xyzw ? 1.0 : 0.0)
```

Set On Greater Or Equal Than (SGE)

Format:

```

20    SGE D[.xyzw],[-]S0[.xyzw],[-]S1[.xyzw]
```

Description:

25 The present instruction set a destination to 1.0/0.0 if source0 is
greater_or_equal/less_than source1.

Operation:

30 Table 8N sets forth an example of operation associated with the SGE
instruction.

Table 8N

```

35      t.x = source0.c***;
      t.y = source0.*c**;
```

```

      t.z = source0.**c*;
```

```

5      t.w = source0.***c;
      if (negate0) {
          t.x = -t.x;
          t.y = -t.y;
          t.z = -t.z;
          t.w = -t.w;
      }
      u.x = source1.c***;
10     u.y = source1.*c**;
      u.z = source1.**c*;
      u.w = source1.***c;
      if (negate1) {
          u.x = -u.x;
          u.y = -u.y;
15     u.z = -u.z;
          u.w = -u.w;
      }
      if (xmask) destination.x = (t.x >= u.x) ? 1.0 : 0.0;
      if (ymask) destination.y = (t.y >= u.y) ? 1.0 : 0.0;
20     if (zmask) destination.z = (t.z >= u.z) ? 1.0 : 0.0;
      if (wmask) destination.w = (t.w >= u.w) ? 1.0 : 0.0;
```

Examples:

```

25     SGE R4,R3,R7      R4.xyzw = (R3.xyzw >= R7.xyzw ? 1.0 : 0.0)
      SGE R3.xz,R6.w,R4  R3.xz = (R6.w >= R4.xyzw ? 1.0 : 0.0)
```

Exponential Base 2 (EXP)

30 Format:

EXP D[.xyzw],[**-**]S0.[xyzw]

Description:

35

The present instruction performs an exponential base 2 partial support. It generates an approximate answer in dest.z, and allows for a more accurate answer of dest.x*FUNC(dest.y) where FUNC is some user approximation to 2**dest.y (0.0 <= dest.y < 1.0). It also accepts a scalar source0. It should be noted that reduced

40 precision arithmetic is acceptable in evaluating dest.z.

EXP(-Inf) or underflow gives (0.0,0.0,0.0,1.0)

EXP(+Inf) or overflow gives (+Inf,0.0,+Inf,1.0)

Operation:

5

Table 80 sets forth an example of operation associated with the EXP instruction.

Table 80

10

```
t.x = source0.c;  
if (negate0) {  
    t.x = -t.x;  
}  
q.x = 2^floor(t.x);  
q.y = t.x - floor(t.x);  
q.z = q.x * APPX(q.y);  
if (xmask) destination.x = q.x;  
if (ymask) destination.y = q.y;  
if (zmask) destination.z = q.z;  
if (wmask) destination.w = 1.0;
```

15

20

where APPX is an implementation dependent approximation of
exponential
base 2 such that

25

$$| \exp(q.y \cdot \log(2.0)) - \text{APPX}(q.y) | < 1/(2^{11})$$

for all $0 \leq q.y < 1.0$.

30

The expression "2^floor(t.x)" should overflow to +Inf and
underflow
to zero.

35 Examples:

EXP R4,R3.z

Logarithm Base 2 (LOG)

40

Format:

LOG D[.xyzw],[-]S0.[xyzw]

Description:

5 The present instruction performs a logarithm base 2 partial support. It generates an approximate answer in dest.z and allows for a more accurate answer of dest.x+FUNC(dest.y) where FUNC is some user approximation of $\log_2(\text{dest.y})$ ($1.0 \leq \text{dest.y} < 2.0$). It also accepts a scalar source0 of which the sign bit is ignored. Reduced precision arithmetic is acceptable in evaluating dest.z.

10

LOG(0.0) gives (-Inf,1.0,-Inf,1.0)

LOG(Inf) gives (Inf,1.0,Inf,1.0)

Operation:

15

Table 8P sets forth an example of operation associated with the LOG instruction.

Table 8P

20

```
t.x = source0.c;
if (negate0) {
    t.x = -t.x;
}
25 if (fabs(t.x) != 0.0f) {
    if (fabs(t.x) == +Inf) {
        q.x = +Inf;
        q.y = 1.0;
        q.z = +Inf;
30     } else {
        q.x = Exponent(t.x);
        q.y = Mantissa(t.x);
        q.z = q.x + APPX(q.y);
    }
35 } else {
    q.x = -Inf;
    q.y = 1.0;
    q.z = -Inf;
40 }
if (xmask) destination.x = q.x;
if (ymask) destination.y = q.y;
if (zmask) destination.z = q.z;
```

if (wmask) destination.w = 1.0;
where APPX is an implementation dependent approximation of
logarithm
5 base 2 such that
$$| \log(q.y)/\log(2.0) - \text{APPX}(q.y) | < 1/(2^{11})$$

10 for all $1.0 \leq q.y < 2.0$.

Examples:

LOG R4,R3.z

15

Light Coefficients (LIT)

Format:

20 LIT D[.xyzw],[-]S0[.xyzw]

Description:

The present instruction provides lighting partial support. It calculates
25 lighting coefficients from two dot products and a power (which gets clamped to –
128.0<power<128.0). The source vector is:

Source0.x = n*l (unit normal and light vectors)
Source0.y = n*h (unit normal and halfangle vectors)
30 Source0.z is unused
Source0.w = power

Reduced precision arithmetic is acceptable in evaluating dest.z. Allowed
error is equivalent to a power function combining the LOG and EXP instructions
35 (EXP(w*LOG(y))). An implementation may support at least 8 fraction bits in the

power. Note that since 0.0 times anything may be 0.0, taking any base to the power of 0.0 will yield 1.0.

Operation:

5

Table 8Q sets forth an example of operation associated with the LIT instruction.

Table 8Q

10

15

20

25

```
t.x = source0.c***;
t.y = source0.*c**;
t.w = source0.***c;
if (negate0) {
    t.x = -t.x;
    t.y = -t.y;
    t.w = -t.w;
}
if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
else if (t.w > 128-epsilon) t.w = 128-epsilon;
if (t.x < 0.0) t.x = 0.0;
if (t.y < 0.0) t.y = 0.0;
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.x;
if (zmask) destination.z = (t.x > 0.0) ?
EXP(t.w*LOG(t.y)) : 0.0;
if (wmask) destination.w = 1.0;
```

30 Examples:

LIT R4,R3

Floating Point Requirements

35

In one embodiment, all vertex program calculations may be assumed to use IEEE single precision floating-point math with a format of s1e8m23 (one signed bit, 8 bits of exponent, 23 bits of magnitude) or better and the round-to-zero rounding mode. Possible exceptions to this are the RCP, RSQ, LOG, EXP, and LIT instructions.

40

It should be noted that (positive or negative) 0.0 times anything is (positive) 0.0. The RCP and RSQ instructions deliver results accurate to $1.0/(2^{22})$ and the approximate output (the z component) of the EXP and LOG instructions only has to be accurate to $1.0/(2^{11})$. The LIT instruction specular output (the z component) is
5 allowed an error equivalent to the combination of the EXP and LOG combination to implement a power function.

The floor operations used by the ARL and EXP instructions may operate identically. Specifically, the x component result of the EXP instruction exactly matches
10 the integer stored in the address register by the ARL instruction.

Since distance is calculated as $(d^2) * (1/\sqrt{d^2})$, 0.0 multiplied by anything is 0.0. This affects the MUL, MAD, DP3, DP4, DST, and LIT instructions. Because if/then/else conditional evaluation is done by multiplying by 1.0 or 0.0 and
15 adding, the floating point computations may require:

$0.0 * x = 0.0$ for all x (including +Inf, -Inf, +NaN, and -NaN)
 $1.0 * x = x$ for all x (including +Inf and -Inf)
 $0.0 + x = x$ for all x (including +Inf and -Inf)

20

Including +Inf, -Inf, +NaN, and -NaN when applying the above three rules is recommended but not required. (The recommended inclusion of +Inf, -Inf, +NaN, and -NaN when applying the first rule is inconsistent with IEEE floating-point requirements.)

25

No floating-point exceptions or interrupts are necessarily generated. Denorms may not necessarily be supported. If a denorm is input, it is treated as 0.0 (ie, denorms are flushed to zero).

30

Computations involving +NaN or -NaN generate +NaN, except for the recommendation that zero times +NaN or -NaN may always be zero. (This exception is inconsistent with IEEE floating-point requirements).

Programming Examples

A plurality of program examples will now be set forth in Table 9.

5

Table 9

The #define statements are meant for a cpp run.

10

Example 1

```
%!VS1.0
; Absolute Value R4 = abs(R0)
```

15

```
MAX R4,R0,-R0;
```

Example 2

20

```
%!VS1.0
; Cross Product | i    j    k | into R2
;               |R0.x R0.y R0.z|
;               |R1.x R1.y R1.z|
```

25

```
MUL R2,R0.zxyw,R1.yzxw;
MAD R2,R0.yzxw,R1.zxyw,-R2;
```

30

Example 3

```
%!VS1.0
; Determinant |R0.x R0.y R0.z| into R3
;             |R1.x R1.y R1.z|
;             |R2.x R2.y R2.z|
```

35

```
MUL R3,R1.zxyw,R2.yzxw;
MAD R3,R1.yzxw,R2.zxyw,-R3;
```

```
DP3 R3,R0,R3;
```

Example 4

5

```
%!VS1.0  
; R2 = matrix[3][3]*v->onrm ,normalize and calculate  
distance vector R3
```

10

```
#define INRM 11;    source normal  
#define N0 16;     inverse transpose modelview row 0  
#define N4 17;     inverse transpose modelview row 1  
#define N8 18;     inverse transpose modelview row 2
```

15

```
DP3 R2.x,v[INRM],c[N0];  
DP3 R2.y,v[INRM],c[N4];  
DP3 R2.z,v[INRM],c[N8];  
DP3 R2.w,R2,R2;
```

20

```
RSQ R11.x,R2.w;  
MUL R2.xyz,R2,R11.x;  
DST R3,R2.w,R11.x;
```

Example 5

25

```
%!VS1.0  
; reduce R1 to fundamental period
```

30

```
#define PERIOD 70;    location PERIOD is  
1.0/(2*PI),2*PI,0.0,0.0
```

```
MUL R0,R1,c[PERIOD].x;    divide by period  
EXP R4,R0;  
MUL R2,R4.y,c[PERIOD].y; multiply by period
```

35

Example 6

40

```
%!VS1.0  
; matrix[4][4]*v->opos with homogeneous divide
```

```
5      #define IPOS    0;    source position
      #define M0      20;    modelview row 0
      #define M4      21;    modelview row 1
      #define M8      22;    modelview row 2
      #define M12     23;    modelview row 3
```

```
10      DP4   R5.w,v[IPOS],c[M12];
      DP4   R5.x,v[IPOS],c[M0];
      DP4   R5.y,v[IPOS],c[M4];
      DP4   R5.z,v[IPOS],c[M8];
      RCP   R11,R5.w;
      MUL   R5,R5,R11;
```

15 Example 7

```
      %!VS1.0
      ; R4 = v->weight.x*R2 + (1.0-v->weight.x)*R3
```

```
20      #define IWGT 11;    source weight
```

```
      ADD   R4,R2,-R3;
      MAD   R4,v[IWGT].x,R4,R3;
```

25 Example 8

```
30      %!VS1.0
      ; output transformed position, xform normal/normalize,
      output two textures
```

```
35      #define IPOS    0;    source position
      #define INORM    11;    source normal
      #define ITEX0    3;    source texture 0
      #define ITEX1    4;    source texture 1
      #define OTEX0    3;    destination texture 0
      #define OTEX1    4;    destination texture 1
      #define N0       16;    inverse transpose modelview row 0
      #define N4       17;    inverse transpose modelview row 1
40      #define N8       18;    inverse transpose modelview row 2
      #define C0       24;    composite row 0
```

```
#define C4      25;    composite row 1
#define C8      26;    composite row 2
#define C12     27;    composite row 3
```

```
5          DP3  R2.x,v[INORM],c[N0];
          DP3  R2.y,v[INORM],c[N4];
          DP3  R2.z,v[INORM],c[N8];
          MOV  o[OTEX0],v[ITEX0];
          DP3  R2.w,R2,R2;
10         RSQ  R2.w,R2.w;
          MUL  R2,R2,R2.w;          keep for later work
          MOV  o[OTEX1],v[ITEX1];
          DP4  o[HPOS].w,v[IPOS],c[C12];
          DP4  o[HPOS].x,v[IPOS],c[C0];
15         DP4  o[HPOS].y,v[IPOS],c[C4];
          DP4  o[HPOS].z,v[IPOS],c[C8];
```

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not
20 limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.